Classification Level: Top secret ( ) Secret ( ) Internal ( ) Public (√)

RKNN_Toolkit Custom Operator Developer Guide

(Technology Department, Graphic Display Platform Center)

Mark: [ ] Editing [√] Released	Version	V0.1.0
	Author	Yang Huacong
	Completed Date	2019-08-22
	Reviewer	Zhuo Hongtian
	Reviewed Date	2019-08-22

福州瑞芯微电子股份有限公司

Fuzhou Rockchips Semiconductor Co . , Ltd

Revision History

Version	Modifier	Date	Modify description	Reviewer
v0.1.0	Yang Huacong	2019-08-22	Initial version	Zhuo Hongtian

Table of Contents

1 Overview

If the model contains operators that are not supported by RKNN-Toolkit, it will fail during the model conversion phase. At this time, you can add unsupported operators through the custom operator function, so that the model can be successfully converted and run.

The current custom operator function is still experimental, and subsequent interfaces may be adjusted. Currently, it can only be used on Linux x64 platforms, and only supports TensorFlow models.

2 Requirements/Dependencies

1) Operating system: Linux x64

2) Dependent software

a) RKNN-Toolkit version 1.2.0 and above.

b) gcc and gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu.

3) Target device driver dependency

a) RK1808

The version of librknn_runtime of the RK1808 platform requires 1.2.0 or higher. You can enter the following command in the shell of rk1808 to view:

```
$ strings /usr/lib64/librknn_runtime.so | grep "librknn_runtime version"
```

b) RK3399Pro

The RK3399Pro platform DRV version needs to be 0.9.9 or more. When running the application, the following log will be printed. You can use the DRV version in the log to determine whether the requirements are met.

```
====================================================
RKNN VERSION:
    API: 0.9.9 (a949908 build: 2019-08-22 22:20:52)
    DRV: 0.9.9 (c12de8a build: 2019-08-22 20:10:17)
====================================================
```

3 User Guide

3.1 RKNN-Toolkit Custom Operator Manual

3.1.1 Create Custom Operator

1) Custom Operator Configuration File

The custom operator is defined by a configuration file in YAML format. The following is an example of the custom operator with RKNN-Toolkit (The configuration file is located: example/custom_op/rknn_custom_op_resize/resize_area.yml).

```
name: ResizeArea
framework: tensorflow
inputs:
  input:
    type: VX_TYPE_TENSOR
outputs:
  output:
    type: VX_TYPE_TENSOR
params:
  size:
    type: VX_TYPE_ARRAY
  align_corners:
    type: VX_TYPE_BOOL
```

The contents of the YAML configuration are as follows:

- Name: Operator name, which must be the same as the operator name of the original model.
- framework: Original model framework, currently only supports tensorflow.
- inputs: Define the operator input. The name of each input needs to be different. The configuration of each input only needs to fill in the type.
- outputs: Define the operator output. The name of each output needs to be different. The configuration of each input only needs to fill in the type.
- params: Define operator parameters. The name of each parameter needs to be different. For the configuration of each parameter item, you only need to fill in the type (type supports

VX_TYPE_ARRAY and scalar types, For scalar types, see the "Read Parameters" section).

Developers can refer to the above configuration to write their own custom operator configuration.

2) Generate custom operator code

After writing the configuration file of the custom operator, the developer can use the following command to generate the code of the custom operator.

```
    cd example/custom_op/rknn_custom_op_resize
    python3 -m rknn.bin.custom_op --action create --config ./resize_area.yml
--op_path ./resize_area
```

After the execution is completed, the operator code will be generated in the resize_area directory.

The parameters of the rknn.bin.custom_op command are as follows:

- **--action/-a**: Pass in "create" to perform the operation of creating the operator code; pass in "build" to perform the operation of compiling the operator code;
- **--config/-c**: The path of the operator configuration file;
- **--op_path/-p**: Path to store the operator code.

3.1.2 Write Custom Operator Code

The code listing of a custom operator is shown below:

```
    resize_area
     ├── makefile.linux.aarch64
     ├── makefile.linux.x64
     ├── op.yml
     ├── rknn_kernel_resizearea.c
     ├── rknn_kernel_resizearea.vx
     └── rknn_op_resizearea.py
```

The code that developers need to complete is mainly:

1) rknn_op_resizearea.py

This Python code is mainly used to obtain the op parameters, calculate the output shape, and define the calculation of the output Tensor during model conversion. For details, please refer to the chapter "Write Custom Operator Python Code" in this document.

2) rknn_kernel_resizearea.c

The C code mainly includes the kernel initialization related callbacks and the kernel's kernel functions. Initialization callback function is using for checking and configuring Kernel parameters。For the writing of the Kernel function of the CPU, please refer to the "Write CPU Kernel" chapter in this document..

3) rknn_kernel_resizearea.vx

It is possible to speed up using the PPU module in the NPU by writing a VX Kernel, 详 For details, please refer to the "Writing VX Kernel" chapter in this document.

4) op.yml

In addition to the operator configuration defined by the developer, the op.yml file also adds several configurations. The configuration item that developers may need to modify is kernel_index, which is used to select which kernel to execute (That is, which kernel in vx_kernel_ResizeArea_list in rknn_kernel_resizearea.c code).

```
name: ResizeArea
framework: tensorflow
inputs:
  input:
    type: VX_TYPE_TENSOR
outputs:
  output:
    type: VX_TYPE_TENSOR
params:
  size:
    type: VX_TYPE_ARRAY
  align_corners:
    type: VX_TYPE_BOOL
out_binary: ResizeArea.rknnop
op_py_file: rknn_op_resizearea.py
vx_file: rknn_kernel_resizearea.vx
kernel_index: 0
```

3.1.3 Compile Custom Operator

3.1.3.1 Compilation Environment Preparation

1) GCC Compiler Installation

Users can install GCC compiler directly through system package management:

```
sudo apt-get install gcc
```

2) Cross Compiler Installation

When compiling for the first time, RKNN-Toolkit will check if a cross-compiler is installed, if not, it will be downloaded automatically from the Internet. Users can also manually download the cross compiler and install it to the specified path, as follows:

```
    wget
https://releases.linaro.org/components/toolchain/binaries/6.3-2017.05/aarch64-linux-gnu/gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu.tar.xz
    mkdir ~/.rknn
    tar xvJf gcc-linaro-6.3.1-2017.05-x86_64_aarch64-linux-gnu.tar.xz -C ~/.rknn/
```

3.1.3.2 Compile Operator Code

Run the following command to compile the code of the custom operator. If the code is wrong, it will abort the compilation and prompt an error. When the compilation is completed successfully, a .rknnop file will be generated in the directory of the operator.

```
python3 -m rknn.bin.custom_op --action build --op_path ./resize_area
```

3.1.4 Register Custom Operator

Only need to call the register_op method before the model conversion and pass in the compiled rknnop file path to register the operator. Each operator needs to call register_op to register separately, the reference code is as follows:

```
rknn.register_op('./resize_area/ResizeArea.rknnop')

rknn.load_tensorflow(…)
```

3.2 Write Custom Operator Python Code

The user needs to complete several method codes in the generated python code, so that RKNN-Toolkit can obtain enough information to complete the conversion and compile the RKNN model. Next we will introduce the Python methods that need to be written.

3.2.1 load_params_from_tf function

This function will be called back during the model conversion phase to get the operator parameters from the TensorFlow node object. The function is defined as follows:

Function	load_params_from_tf
Description	Get operator parameters.
Parameters	node_def: The parameter is a tf.NodeDef object, including information about the original model corresponding to the node.
	tensor_data_map: Includes all input constants Tensor of this node。
Return	dict: Parameter dictionary. The key of each parameter must be consistent with the parameter name specified in the YAML configuration.

The sample code is as follows:

```
def load_params_from_tf(self, node_def, tensor_data_map):
    p = dict()
    # set params dict
    p['size'] = tensor_data_map['C:out0'].tolist()
    p['align_corners'] = node_def.attr['align_corners'].b
    return p
```

3.2.2 compute_output_shape function

This function will be called back during the model conversion phase to obtain the shape information of the operator output Tensor. The function is defined as follows:

Function	compute_output_shape
Description	Get the shape of the operator output Tensor.
Parameters	inputs_shape: The shape of all the inputs of this op.
	Params: The parameters of this node.
Return	List: Output a list of Tensor Shapes.

The sample code is as follows:

```
def compute_output_shape(self, inputs_shape, params):
    outputs_shape = [Shape() for i in range(len(self.def_output))]
    # set outputs shape by set_shape()
    in_shape = inputs_shape[0].format('nhwc')
    in_channel = in_shape[-1]
    out_shape = []
    out_shape.extend(params['size'])
    out_shape.append(in_channel)
    outputs_shape[0].set_shape(shape=out_shape, fmt='nhwc')
    return outputs_shape
```

3.2.3 compute_output_tensor function

The model quantization phase needs to call back this function to get the calculated output of op Tensor. In this method, developers can define operator calculation methods by calling TensorFlow functions or by tf.py_func and Numpy. The function is defined as follows:

Function	compute_output_tensor
Description	Define the calculation method of the operator output Tensor.
Parameters	const_tensor: Input constant tensor for this node.
	inputs_tensor: The input tensor list of this node, each element of the list is a tf.Tensor object。

	Params: The parameters of this node.
Return	list: Output a list of Tensors (of type tf.Tensor).

The sample code is as follows:

```
def compute_output_tensor(self, const_tensor, inputs_tensor, params):
    outputs_tensor = list()
    # compute outputs tensor
    out = tf.image.resize_area(inputs_tensor[0],
                                        size=params['size'],
                                       align_corners=params['align_corners'])
    outputs_tensor.append(out)
    return outputs_tensor
```

3.3 Write CPU Kernel

Developers can implement the Kernel function of the CPU version of the operator first, which makes it easier to write and verify that the results are correct. The cpu_kernel_function function of the generated C code file is a CPU Kernel function. When the network executes to this layer node, the function will be called back. The function is defined as follows:

Function	cpu_kernel_function
Description	Define the calculation method of the operator output Tensor.
Parameters	vx_node node: This node.
	vx_reference* parameters: Store input tensor, output tensor, and parameters separately in order.
	uint32_t paramNum: Number of parameters.
Return	vx_status: reference vx_status_e.

The developer first needs to read the input Tensor data from the function parameter vx_reference * parameter. then write the operator calculation method code. Finally write the output data to the output tensor. The following will introduce each one in detail.

3.3.1 Read and Write Tensor

The function parameter vx_reference * parameter of cpu_kernel_function stores the input Tensor, output Tensor, and operator parameters in the order defined by the variable "kernel_params" in the C code. Take the definition of "kernel_params" as an example: parameter [0] is the input tensor; parameter [1] is the output tensor; parameter [2] and parameter [3] are operator parameters.

```
static vx_param_description_t kernel_params[] =
    {
        {VX_INPUT, VX_TYPE_TENSOR, VX_PARAMETER_STATE_REQUIRED},
        {VX_OUTPUT, VX_TYPE_TENSOR, VX_PARAMETER_STATE_REQUIRED},
        {VX_INPUT, VX_TYPE_ARRAY, VX_PARAMETER_STATE_REQUIRED},
        {VX_INPUT, VX_TYPE_SCALAR, VX_PARAMETER_STATE_REQUIRED}
    };
```

In OpenVX, vx_reference (refer to the OpenVX manual Object: vx_reference) is a general reference that can be directly converted to other types, such as vx_tensor, vx_array, vx_scalar, etc. Because the input Tensor is of type vx_tensor, vx_reference can be directly converted to vx_tensor here.

OpenVX provides an interface to query and read and write vx_tensor objects (Refer to OpenVX manual Object:Tensor).

- **vxQueryTensor**: Querying Tensor attributes (supports querying Tensor dimensions, number of dimensions, data types, etc.);
- **vxCopyTensorPatch**: Copy data to / from Tensor.

The following is the sample code of vx_tensor (for the complete code, please refer to the example/custom_op in RKNN_Toolkit)：

```
vx_status status;
uint32_t dim_num；
uint32_t dims[4] = {0};
// Read Tensor Attributes
status = vxQueryTensor(tensor, VX_TENSOR_NUMBER_OF_DIMS,
                       &dim_num, sizeof(uint32_t));
status = vxQueryTensor(tensor, VX_TENSOR_DIMS,
                       size, sizeof(uint32_t) * dim_num);
......
// Read Tensor Data
```

```
status = vxCopyTensorPatch((vx_tensor)parameters[0],
                           dim_num, view_start, view_end, stride_size,
                           src_buffer, VX_READ_ONLY, 0);
// Write Data to Tensor
status = vxCopyTensorPatch((vx_tensor)parameters[1],
                           dim_num, view_start, view_end, stride_size,
                           dst_buffer, VX_WRITE_ONLY, 0);
```

Note that the view_start, view_end, and user_stride parameters in the parameters of vxCopyTensorPatch.

- **view_start**: This parameter indicates the starting position of the Tensor to be read/written (type is vx_size []);
- **view_end**: This parameter indicates the end position of the Tensor to be read / written (type is vx_size []);
- **stride_size**: This parameter indicates the span of the read / write Tensor (type is vx_size []). Each element of the array needs to be assigned as follows:

```
stride_size[0] = sizeof(element dtype)
stride_size[1] = dims[0] * stride_size[1]
stride_size[2] = dims[1] * stride_size[2]
stride_size[3] = dims[2] * stride_size[3]
```

3.3.2 Read Parameters

Currently, the parameters of custom operators support scalar (VX_TYPE_SCALAR) and array (VX_ARRAY). Like vx_tensor, you can directly convert vx_reference to vx_scalar or vx_array according to the definition.s

1） Read vx_scalar Type Parameter

vx_scalar represents a scalar type (Refer to OpenVX manual Object:Scalar), The types included are as follows:

Primitive Type	OpenVX Type	vx_type_e Enum
char	vx_char	VX_TYPE_CHAR
signed char	vx_int8	VX_TYPE_INT8

unsigned char	vx_uint8	VX_TYPE_UINT8
short	vx_int16	VX_TYPE_INT16
unsigned short	vx_uint16	VX_TYPE_UINT16
int	vx_int32	VX_TYPE_INT32
unsigned int	vx_uint32	VX_TYPE_UINT32
long long	vx_int64	VX_TYPE_INT64
unsigned long long	vx_uint64	VX_TYPE_UINT64
	vx_float16	VX_TYPE_FLOAT16
float	vx_float32	VX_TYPE_FLOAT32
double	vx_float64	VX_TYPE_FLOAT64
int	vx_enum	VX_TYPE_ENUM
unsigned int	vx_size	VX_TYPE_SIZE
	vx_bool	VX_TYPE_BOOL

OpenVX provides an interface to query and read and write vx_scalar objects:

- **vxCreateScalar**: Create a vx_scalar object;
- **vxCreateScalarWithSize**: Create vx_scalar object (with object size parameter);
- **vxReleaseScalar**: Release the vx_scalar object;
- **vxQueryScalar**: Query vx_scalar object (supports querying vx_scalar type);
- **vxCopyScalar**: Read and write vx_scalar objects;
- **vxCopyScalarWithSize**: Read and write vx_scalar objects (with object size parameters).

The following is sample code for vx_scalar:

```
vx_enum scalar_type;
vx_bool align_corners = 0;
vxQueryScalar((vx_scalar)parameters[3], VX_SCALAR_TYPE,
                &scalar_type, sizeof(vx_enum));
vxCopyScalar((vx_scalar)parameters[3], &align_corners,
                VX_READ_ONLY , VX_MEMORY_TYPE_HOST);
```

2） Read vx_array Type Parameter

OpenVX uses vx_array type to represent array (Refer to the OpenVX manual Object:Array), The corresponding type enumeration value is VX_TYPE_ARRAY. OpenVX also provides a series of functions to query and read and write vx_array objects:

- **vxCreateArray**: Create a vx_array object;
- **vxReleaseArray**: Release vx_array object;
- **vxQueryArray**: Query the vx_array object (supports querying the type of array items, the number of array items, the capacity of the array, and the size of the array items);
- **vxMapArrayRange**: Map a range of vx_array objects for user access.s

Following is the sample code for vx_array:

```
int32_t size[2];
vx_enum array_item_type;
vx_size array_item_num;

vxQueryArray((vx_array)parameters[2], VX_ARRAY_ITEMTYPE,
              &array_item_type, sizeof(vx_enum));
vxQueryArray((vx_array)parameters[2], VX_ARRAY_NUMITEMS,
              &array_item_num, sizeof(vx_size));

vxCopyArrayRange((vx_array)parameters[2], 0, 2, sizeof(int32_t), (void *)size,
                     VX_READ_ONLY, VX_MEMORY_TYPE_HOST);
```

Note that the item_type of the vx_array of the current custom operator is of type int32_t.

3.3.3 Write Operator Implementation Code

After obtaining the data and parameters of the operator input Tensor, the user can write the implementation code of the operator. When using input Tensor data, pay attention to the input Tensor data type of the operator (Because the custom operator is not quantized, the input Tensor data type is float16) and the order of arrangement (default is NCHW).

Developers can also call interface functions provided by OpenVX (Refer to the OpenVX manual Vision Function), It should be noted that functions starting with vxu can be directly called to get the result.

3.4 Write VX Kernel

Developers can use the PPU module in the NPU to speed up by writing VX Kernel code. The C code in the generated code contains callback functions such as initialization, de-initialization, and parameter verification of the VX kernel; files with the vx extension are kernel functions.

Note that writing VX Kernel requires some OpenCL development experience.

3.4.1 Kernel Initialization

In the initialization function, developers need to configure Kernel's execution parameters according to their needs. VX Kernel execution parameters are similar to OpenCL's NDRange, and both have work-item and work-group concepts:

- **work-item**: Worker thread, each Work-item has a unique global ID (3 dimensions [x, y, z]).
- **work-group**: It is composed of one or more work-items and is the basic unit of thread switching.

The definition of VX Kernel execution parameters is shown below:

```
typedef struct _vx_kernel_execution_parameters {
      vx_uint32 workDim;
      vx_size globalWorkOffset[VX_MAX_WORK_ITEM_DIMENSIONS];
      vx_size globalWorkScale[VX_MAX_WORK_ITEM_DIMENSIONS];
      vx_size localWorkSize[VX_MAX_WORK_ITEM_DIMENSIONS];
      vx_size globalWorkSize[VX_MAX_WORK_ITEM_DIMENSIONS];
} vx_kernel_execution_parameters_t;
```

Parameter	Type	Description
workDim	vx_uint32	Work-item dimensions, valid values are 1, 2, and 3.
GlobalWorkOffset[i]	vx_size	The initial offset of the global ID of work-item [i].
globalWorkScale[i]	vx_size	The step value of the global ID of work-item [i].
localWorkSize[i]	vx_size	The size of the work-group. That is, how many work-items a work-group contains.
globalWorkSize[i]	vx_size	The total work-item size (globalWorkSize must be an integer multiple of localWorkSize).

3.4.2 Read Data From Tensor

Reading input Tensor data in VX Kernel can use VXC_ReadImage2DArray function.

Function	VXC_ReadImage2DArray(Dest, Image, Coord, Offset, Info)
Description	This function can read up to 128bit data from the image, and can also be used to read data from a 3-dimensional Tensor.
Parameters	Dest: Read data storage location.
	Image: Image (Tensor) to be read.
	Coord: Coordinate to be read (int4).
	Offset: XY offset position can be set using the following macro function: VXC_5BITOFFSET_XY(offsetX, offsetY) The offsetX and offsetY values range from -16 to 15.
	Info: Control information, which can be set using the following macro functions: VXC_MODIFIER (StartBin, EndBin, SourceBin, RoundingMode, Clamp) ● StartBin: The starting position of the target storage ● EndBin: End position of target storage ● SourceBin: Source starting position ● RoundingMode: Rounding mode, the values that can be set are (VXC_RM_TowardZero: round down; VXC_RM_TowardInf: round up; VXC_RM_ToNearestEven: recent round) ● Clamp: Whether to use Clamp for data reduction when the target type is small. 0 is truncated; 1 is Clamp.
Function	void

The sample code is as follows:

```
int4 Coord0 = int4(0,0,0,0);
VXC_ReadImage(Dst0, Image, Coord0, VXC_5BITOFFSET_XY(0,0),
                    VXC_MODIFIER(0, 15, 0, VXC_RM_TowardZero, 0));
```

Assuming that the input Image is a 3-dimensional Tensor (NCHW), the above code will read 16 Dest0 data from the first line of the first channel.

3.4.3 Write Data To Tensor

Reading the input Tensor data in VX Kernel can use the VXC_WriteImage2DArray function.

Function	VXC_WriteImage2DArray (Image, Coord, Color, Info)
Description	This function can write up to 128bit data to an image, and it can also write data to a 3D Tensor.
Parameters	Image: Image (Tensor) object to be written.
	Coord: The coordinate to be written (int4).
	Color: Data to be written.
	Info: Control information, which can be set using the following macro functions: VXC_MODIFIER (StartBin, EndBin, SourceBin, RoundingMode, Clamp) ● StartBin: The starting position of the target storage ● EndBin: End position of target storage ● SourceBin: Source starting position ● RoundingMode: Rounding mode, the values that can be set are (VXC_RM_TowardZero: round down; VXC_RM_TowardInf: round up; VXC_RM_ToNearestEven: recent round) ● Clamp: Whether to use Clamp for data reduction when the target type is small. 0 is truncated; 1 is Clamp.
Return	void

The sample code is as follows:

```
int4 Coord0 = int4(0,0,0,0);
VXC_WriteImage(Image, Coord0, Val0,
                    VXC_MODIFIER(0, 15, 0, VXC_RM_TowardZero, 0));
```

Assuming the output Image is a 3-dimensional Tensor (NCHW), then the above code writes 16 Val0 type data to the first line of the first channel.

3.4.4 Write Operator Implementation Code

Except for the read and write functions of Tensor, the syntax of VX Kernel is basically the same as that of OpenCL1.2 Kernel. Developers can refer to OpenCL1.2 Official guide to write the implementation code of the operator. When using input Tensor data, you need to pay attention to the input Tensor data type of the operator (because custom operators are not quantized, so the input Tensor data type is float16) and the order of arrangement (default is NCHW).